Chapter FOUR
Interfaces

Exam Objectives

Develop code that declares, implements and/or extends interfaces and use the atOverride annotation.

What's an interface?

The first time you look an interface, you'll probably think that it's like a class with just methods definitions:

```
interface Monitorable {
    void monitor();
}
```

And for practical terms, you're right.

An interface is a data type that just defines (abstract) methods that one class must implement.

Although conceptually, it's more interesting than that, because this allows you to define what a class can do without saying how to do it. That's why it's said that an interface is a contract.

Any class that implements an interface must provide an implementation for all the methods of the interface, otherwise, the class has to be marked as `abstract`.

As a class is defined with the `class` keyword, an interface is defined with the `interface` keyword.

If a class wants to implement an interface, it has to specify it with the `implements` keyword.

Defining an interface

```
public interface Monitorable {
    public static final int ID = 0;
    public abstract void monitor();
}
```

Implementing an interface

```
class Server implements Monitorable {
    public void monitor() {
        // Implementation code
    }
}
```

Like a class, an interface has either `public` or default accessibility:

```
public interface PublicAccessInterface {
    // ...
}
```

```
interface DefaultAccessInterface {
    // ...
}
```

Interfaces are `abstract` by default (you don't have to specify it):

```
public abstract interface PublicAccessInterface {
    // This is the same as the definition above
}
```

This means two things:

- You can't instantiate an interface directly, it has to be implemented by a class to use it.
- An interface cannot be marked as `final` .

The methods defined in an interface are by default **PUBLIC** and **ABSTRACT**, the compiler will treat them as such even if you don't specify it.

So even though you define an interface like this:

```
interface Monitorable {
    void monitor();
    void setup();
}
```

For the compiler, the interface will look like this:

```
interface Monitorable {
    public abstract void monitor();
    public abstract void setup();
}
```

That's the reason you must mark the method as `public` when it's implemented:

```
class Server implements Monitorable {
    public void monitor() {
        // Implementation
    }
    public void setup() {
        // Implementation
    }
}
```

Also, that's the reason that, if one or more of the interface's methods are not implemented, you must mark the class (and the methods) as `abstract`:

```
abstract class Server implements Monitorable {
    public void monitor() {
        // Implementation
    }
    public abstract void setup();
}
```

Fields declared in an interface are by default **PUBLIC**, **STATIC**, and **FINAL**. Like methods, the compiler will treat them as such even if you don't specify it.

This means that fields are **CONSTANTS** instead of **VARIABLES**:

```
interface Monitorable {
    int ID = 0; // You have to assign a value at creation time
}
class Resource implements Monitorable {
    void change() {
        ID = 5; // This WON'T compile
    }
}
```

This also means that the following declarations are all equivalent inside an interface:

```
int ID = 0;
public int ID = 0;
static int ID = 0;
final int ID = 0;
public static ID = 0;
public final ID = 0;
static final ID = 0;
public static final ID = 0;
```

So watch out for declarations that won't compile, like these:

```
interface Monitorable {
    private int ID = 0; // It cannot be private
    int TIMEOUT; // It's final, so you have to provide a value
}
```

There are two rules regarding inheritances and interfaces:

- A class can implement (not extend from) any number of interfaces.
- An interface can extend any number of interfaces, but it cannot extend from a class.

Implementing an interface is a type of inheritance. When a class implements an interface, we can use it like this to take advantage of polymorphism:

```
interface Monitorable {
    void monitor();
}
class Disk implements Monitorable {
    public void monitor() {
        System.out.println("Monitoring Disk");
    }
}
class Server implements Monitorable {
    public void monitor() {
        System.out.println("Monitoring Server");
```

```
    }
}
public class Test {
    public static void main(String args[]) {
        Monitorable m = new Disk();
        m.monitor();
        m = new Server(); // Change implementation
        m.monitor();
    }
}
```

This is the output:

```
Monitoring Disk
Monitoring Server
```

A class cannot extend from more than one class, but it can implement more than one interface. You can see the reason with an example. Consider:

```
class Truck {
    public void accelerate() {
        System.out.println("Accelerating truck...");
    }
}
class CompactCar {
    public void accelerate() {
        System.out.println("Accelerating compact car...");
    }
}
```

If you could inherit from multiple classes:

```
class Car extends Truck, CompactCar {
    public void run() {
        accelerate();
        // ...
    }
}
```

Which `accelerate()` method would Java choose?

This is a problem that the designers of Java decided to avoid by not allowing multiple inheritance.

But what if we were using interfaces?

```
interface Truck {
    void accelerate();
}
interface CompactCar {
    void accelerate();
}
class Car implements Truck, CompactCar {
    public void accelerate() {
        System.out.println("Accelerating car"); }
        // ...
}
```

As the interfaces don't provide an implementation, there's only one (`Car` 's implementation) so there's no conflict and we avoid the problem altogether!

And if the methods have the same name but different parameters:

```
interface Truck {
    void accelerate();
}
interface CompactCar {
    void accelerate(int speed);
}
```

They're considered two different methods because the method's signature is different, so the implementing class has to implement both versions of the methods:

```
class Car implements Truck, CompactCar {
    public void accelerate() {
        // implementation
    }
    public void accelerate(int speed) {
        // implementation
    }
}
```

However, when the methods only differ in their return type, since the return type is not considered in the method's signature, the Java compiler will generate an error:

```
interface Truck {
    void accelerate();
}
interface CompactCar {
    int accelerate();
}
class Car implements Truck, CompactCar {
    // Java will complain about duplicate methods
    // and incompatible return types
    public void accelerate() {
        // implementation
    }
    public int accelerate() {
        // implementation
    }
}
```

Optionally, to make things clearer, we can use the `@Override` annotation.

```
interface CompactCar {
    int accelerate();
}
class Car implements CompactCar {
    @Override
    public int accelerate() {
        // implementation
    }
}
```

`@Override` indicates that a method overrides a method declaration in a supertype, either in an interface or a parent class.

If the annotated method doesn't override or implement the method correctly, the compiler will generate an error.

This is the only function of the annotation. It's useful in cases like when there's a not-so-obvious error caused by a typo:

```
interface CompactCar {
    int accelerate();
```

```
}
class Car implements CompactCar {
    // Compiler will mark an error about method
    // "acelerate" not overriding or implementing something
    @Override
    public int acelerate() {
        // implementation
    }
}
```

Finally, an interface can only extend other interfaces.

```
interface Monitorable {
    void monitor();
}
interface Pluggable {
    void plug();
}
interface Resource extends Monitorable, Pluggable {
    void printInfo();
}
```

A non-abstract class implementing `Resource`, must implement all the methods of the three interfaces:

```
class Disk implements Resource {
    public void monitor() {
        // implementation
    }
    public void plug() {
        // implementation
    }
    public void printInfo() {
        // implementation
    }
}
```

What's new in Java 8?

Assume we have an interface like this:

```
interface Processable {
    void processInSequence();
}
```

And an implementation:

```
class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
}
```

We know that when a class implements an interface, unless the class is marked as `abstract`, it has to implement **ALL** the methods of that interface.

So if we add another method to `Processable`, for example:

```
interface Processable {
    void processInSequence();
```

```
    void processInParallel();
}
```

We have to update the class to avoid a compilation error:

```
class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Processing in parallel");
    }
}
```

That was easy. But think about the following:

- What if we have hundreds of classes implementing `Processable` ?
- What if we can't update or don't have access to the code for some reason?
- What if the new method is not needed or doesn't make sense for some implementations?

These are real problems, sometimes not easy to solve.

However, Java 8 gives us *default* methods. We don't have to provide implementations for them because they are non-abstract methods.

In other words, interfaces now allow methods with a **BODY**. And this is not as simple as it sounds.

Default methods

The main reason for adding *default* methods to interfaces was to support *interface evolution*, to add new functionality to interfaces and at the same time, ensuring compatibility with the code written for older versions.

There are two other side effects worth mentioning:

- **We can now design *optional* methods**. We can have methods with limited or default functionality so the classes implementing the interfaces can decide if they keep that functionality or provide another one.
- **We can have *utility* methods directly on the interface**. Methods that get or create resources, for example, made just for convenience and possibly implemented in terms of non-default methods of the interface.

However, now that interfaces can provide behavior, the difference with abstract classes is not very clear in some cases. There are still two significant differences:

- A class can only extend from **ONE** abstract class, but it can implement **MULTIPLE** interfaces.
- An abstract class can have a state through **INSTANCE** variables (fields). An interface **CAN'T**.

Default methods come with many rules, especially regarding inheritance. But let's start with their syntax.

Defining a default method

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        /** Default implementation goes here */
    }
}
```

An interface can have any number of abstract and default methods.

All methods with the keyword default must have a body.

Default methods are public implicitly, just as any other method of an interface.

By making `processInParallel()` a *default* method, the implementing class gets it automatically. Here's the complete example:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processing in parallel");
    }
}
public class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInSequence();
        t.processInParallel(); // It compiles just fine
    }
}
```

This is the most simple scenario, where the implementing class inherits the *default* method.

Before presenting more complex scenarios, let's see what the restrictions are when using *default* methods.

Default methods cannot be final.
If a method is `final`, it cannot be overridden by the implementing classes, which doesn't favor the primary objective of *default* methods.

Default methods cannot be synchronized.
This was a deliberate decision by the designers of the language. If a method is made synchronized in the interface, it would mean that all the implementing classes would inherit this behavior. But this decision should belong to the implementation; the interface has no reasonable basis for assuming what the synchronization policy should be.

Default methods are always public.
Like other methods of an interface. Contrast this with an abstract class, where you can choose the visibility of the method.

You cannot have default methods for the `Object`'s class methods.
An interface cannot provide default implementations for:

```
boolean equals(Object o)
int hashCode()
String toString()
```

If an interface has methods with those signatures, the compiler will throw an error. The reason is that those methods are all about the object's state. Since interfaces do not have a state, these methods should be in the implementing classes.

And now, to the more complex scenarios.

Class overrides default method

Classes always **WIN** over interfaces. If a class overrides a default method, the class method will be the one used. For example:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}
public class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Class parallel");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t. processInParallel();
    }
}
```

The output would be:

```
Class parallel
```

This is true even if the class redefines the default method as `abstract`:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}
// Class Task has to be abstract
abstract class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public abstract void processInParallel();
}
```

If for some reason, you need to call the default implementation of the method, you can do it with the name of the interface followed by the keyword `super`:

```
public void processInParallel() {
    Processable.super.processInParallel();
}
```

This only works with default methods. Calling a non-default method in this way will result in a compilation error. Also, `super` must be used with a direct super interface of the class.

Another scenario related to this rule is when an inherited instance method from a class overrides a default interface method:

```
interface Processable {
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}
class Process {
    public void processInParallel() {
        System.out.println("Class parallel");
    }
}
public class Task
        extends Process implements Processable {
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

The output is:

```
Class parallel
```

The method `processInParallel()` returns the string `"Class parallel"` since the class `Task` inherits the method `processInParallel()` from the class `Process`, which overrides the default method of the same name in the interface `Processable`.

Interface inheritance with default methods

More specific interfaces (or classes) always **WIN** over less specific ones. The default methods of the shallower interfaces in an inheritance hierarchy will be used. For example:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processable parallel");
    }
}
interface Parallelizable extends Processable {
    default void processInParallel() {
        System.out.println("Parallelizable parallel");
    }
}
public class Task implements Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

The output would be:

```
Parallelizable parallel
```

The interface `Parallelizable` inherits the default method `processInParallel()`, but since it redefines the method when `Task` implements it, its implementation is the one called.

If `Parallelizable` defined `processInParallel()` as `abstract`:

```
interface Parallelizable extends Processable {
    abstract void processInParallel();
}
```

Then `Task` would have to implement the method (to not become an `abstract` class):

```
public class Task implements Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Task parallelizable");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

The output would be:

```
Task parallelizable
```

Multiple interface inheritance with default methods

Classes can implement multiple interfaces. What happens when two interfaces have the same default method? Which one does the implementing class will choose? For example:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processable parallel");
    }
}
interface Parallelizable {
    default void processInParallel() {
        System.out.println("Parallelizable parallel");
    }
}
public class Task
            implements Processable, Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

It turns out that the result is a compiler error:

```
Duplicate default methods named processInParallel with the parameters () and () are inh
```

The compiler doesn't know which to choose, so it generates an error. In this case, `Task` has to provide an implementation (honoring the previous rule of *more specific interfaces (or classes) always wins over less specific ones)* to override the interface `default` methods and solve the issue:

```
public class Task
            implements Processable, Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Task parallelizable");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

And now, the output would be:

```
Task parallelizable
```

Of course, we can always call a default implementation with:

```
public void processInParallel() {
    Processable.super.processInParallel();
}
```

Defining a static method

```
interface Processable {
    static void log() {
        /** Implementation goes here */
    }
}
```

Static methods in interfaces are defined just like static methods in classes, with the keyword `static` *.*

Static methods are `public` *implicitly, just as any other method of an interface.*

An interface can have any number of static methods.

Static methods

Whenever we refer to something static, we mean something that belongs to a class, not to a particular instance or object of that class. Static methods on interfaces follow the same concept; they belong to the interface where they are declared.

They were added to assist default methods and to better organize helper methods, because generally, helper or utility methods are defined in another class (like

`java.utils.Collections`), instead of where they naturally belong.

For example, the `java.util.Comparator` interface defines the static method:

```
static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)
```

Used by the default method:

```
default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)
```

Interface `static` methods are not inherited, you must prefix the method with the interface name:

```
interface Parallelizable {
    static void log(String s) {
        System.out.println(s);
    }
    default void processInParallel() {
        log("Parallelizable parallel");
    }
}
public class Task implements Parallelizable {
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
        // t.log("The end"); Doesn't compile
        // Task.log("The end"); Doesn't compile either
        Parallelizable.log("The end"); // Compiles!
    }
}
```

The output is:

```
Parallelizable parallel The end
```

Key Points

- An interface is a data type that just defines (`abstract`) methods that one class must implement.
- An interface is defined with the `interface` keyword. If a class wants to implement an interface, it has to specify it with the `implements` keyword.
- An interface has either `public` or `default` accessibility and is abstract by default.
- The methods defined in an interface are by default `public` and `abstract` . The compiler will treat them as such even if you don't specify it.
- Fields declared in an interface are by default `public` , `static` , and `final` . Like methods, the compiler will treat them as such even if you don't specify it.
- This means that fields are constants instead of variables.
- A class can implement (not extend from) any number of interfaces.
- An interface can extend any number of interfaces, but it cannot extend from a class.
- `@Override` indicates that a method overrides a method declaration in a supertype, either in an interface or a parent class.
- If the annotated method doesn't override or implement the method correctly, the compiler will generate an error.
- Java 8 introduced default methods in interfaces to support *interface evolution*, adding new functionality to interfaces and at the same time, ensuring compatibility with the code written for older versions.

- Default methods are methods marked with the `default` keyword, and they must have a body. The implemented classes can use and optionally redefine these methods.
- Default methods are always `public`, but not `static`.
- Default methods cannot be `synchronized` or `final`.
- You cannot define default methods with the same signature as the `Object` class methods:

```
boolean equals(Object o);
int hashCode();
String toString();
```

- In an inheritance hierarchy, the most specific method is the one called.
- One case is if a class redefines a default method, the class method is the one that gets called.
- Another case is if an interface redefines a default method inherited from a super interface, the method of the subinterfaces is the one called.
- If a class implements two different interfaces with the same default method (same method signature), the class must redefine the method. Otherwise, a compiler error is generated.
- If you want to call the default method of the interface from the implementing class (or extending interface), do it this way:

```
NameOfTheInteface.super.defaultMethod();
```

- Java 8 also introduced `static` methods in interfaces so they can contain helper or utility methods.
- Static methods are marked with the `static` keyword, and they also must have a body.
- Static methods in interfaces have the same meaning that `static` methods in classes, so they are not inherited.
- If you want to call a default method from a particular interface, do it this way:

```
NameOfTheInteface.staticMethod();
```

Self Test

1. Given:

```
interface A {
    default int aMethod() {
        return 0;
    }
}
public class Test implements A {
    public long aMethod() {
        return 1;
    }
    public static void main(String args[]) {
        Test t = new Test();
        System.out.println(t.aMethod());
    }
}
```

What is the result?
A. `0`
B. `1`
C. Compilation fails
D. An exception occurs at runtime

2. Given:

```
interface B {
    default static void test() {
        System.out.println("B test");
    }
}
public class Question_4_2 implements B {
    public void test() {
        System.out.println("Q test");
    }
    public static void main(String[] args) {
        Question_4_2 q = new Question_4_2();
        q.test();
    }
}
```

What is the result?
A. `B test`
B. `Q test`
C. Compilation fails
D. An exception occurs at runtime

3. Given:

```
interface C {
    default boolean equals(C obj) {
        return obj == this;
    }
}
public class Question_4_3 implements C {
    public static void main(String[] args) {
        Question_4_3 q = new Question_4_3();
        System.out.println(q.equals(q));
    }
}
```

What is the result?
A. `true`
B. `false`
C. Compilation fails
D. An exception occurs at runtime

4. Given:

```
interface D {
    default void print() {
        System.out.println("D");
    }
}
interface E extends D {
    default void print() {
        System.out.println("E");
    }
}
public class Question_4_4 implements E {
    public void print() {
        E.super.print();
    }
    public static void main(String[] args) {
        Question_4_4 q = new Question_4_4();
        q.print();
   }
}
```

What is the result?
A. `D`
B. `E`
C. `D` and then `E`
D. Compilation fails
E. An exception occurs at runtime

5. Given:

```
interface F {
    static void test() {
        System.out.println("F test");
    }
}
public class Question_4_5 implements F {
    public void test() {
        System.out.println("Q test");
    }
    public static void main(String[] args) {
        F q = new Question_4_5();
        q.test();
    }
}
```

What is the result?
A. `F test`
B. `Q test`
C. Compilation fails
D. An exception occurs at runtime

6. Given:

```
interface G {
    default void doIt() {
        System.out.println("G - Do It");
    }
}
interface H {
    void doIt();
}
public class Question_4_6 implements G, H {
    public void doIt() {
        System.out.println("Do It");
    }
    public static void main(String[] args) {
        Question_4_6 q = new Question_4_6();
        q.doIt();
    }
}
```

What is the result?
A. `G - Do It`
B. `Do It`
C. Compilation fails
D. An exception occurs at runtime

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

03. Inner Classes 05. Enumerations